Object Oriented Programming System

 ( OOPS )

* If you want to develop a software application, you have to use one of the two programming paradigms(models).
1. Procedure Oriented Programming model
2. Object Oriented Programming model.
* In POP model, the applications are developed by creating multiple functions.
* For a small scale application, POP model is suitable.
* But for large scale applications, if many functions are created in the program, then it is difficult to find out which function is modifying which data. So increases complexity.
* So, another programming model called OOP was introduced to develop any scale applications.
* In POP, the data exists at one place and the logic on the data exists at another place. So it kills the readability.
* In OOP, the data and the logic on that data will exist at one place. So it improves the readability.
* C and Pascal are the Procedure Oriented Programming languages.
* Java, Python , … are the Object Oriented Programming languages.
* The principles of Object Oriented Programming System are,
1. abstraction
2. encapsulation
3. inheritance
4. polymorphism
* If a programming language provides a way to implement these principles in an application then it is called Object Oriented programming language.

abstraction:

* It is a process of showing essential information and hiding un-essential information based on the user of a system.
* For example, in a banking system, the users are clerks, accountants and managers.
* The information shown to the clerks is less than the information shown to the accountants and it is less than the information shown to the managers. This is abstraction.
* Suppose, you are calling a method of Java API, you are given with method name, parameters and return type information, but not the logic of the method. This is called abstraction.
* Java allows the programmers/developers to implement abstraction in two ways.
1. using abstract classes
2. using interfaces.

encapsulation:

* It is a process of combining the data and related functionality together at a single place/unit, to protect the data from un-authorized modification.
* we can implement encapsulation in Java, by creating a class with private variables and public getter and setter methods.

 inheritance:

* It is a process of creating new classes from existing classes to achieve re-usability and to avoid redundancy. (duplication of code).
* The newly created class is called child class and the existing class is parent class.
* The newly created class inherits properties and the behavior from the existing class.
* In Java, extends keyword is given to implement inheritance.

polymorphism:

* polymorphism word is constructed from two greek words poly and morphos.
* poly denotes many and morphos denotes forms.
* polymorphism denotes many forms.
* A functionality can have many forms.
* To implement/define a functionality, we create methods.
* For ex, address verification in a banking system is a functionality and it has multiple forms like with adhaar card, or with voter id, etc..
* In Java, we can implement polymorphism using two mechanisms.
1. method overloading
2. method overriding

class and object:

-----------------

 class: A class is a template for a group of objects which shares common attributes and behavior.

 (or)

 A class is a blue print for a group of objects which shares common attributes and behavior.

* class is a word took from classification.
* A class when it is created with a class keyword, contains variables and methods.
* The variables can also be called as data members and methods can also be called member methods.

syntax:

 class <classname> {

 variables;

 methods;

 }

ex:

 class Employee {

 int empno;

 String ename;

 double sal;

 void calculateBonus() {

 //logic

 }

 void display() {

 //logic

 }

 }

* an object is an instance of a class.
* an object represents a real world entity.
* In object oriented programming model, everything and anything is considered as an object.
* An application can use different categories of objects. So we create different templates in an application, which are called classes.
* In Java, we have to use new keyword for creating an object.

syntax:

 classname referencevariable = new classname();

 1 2 3 4

1. classname : The name of the class for which

 we are creating an object.

1. referencevariable : It is the name given to an object. It is also called object name.
2. new : It is a keyword, which allocates memory for an object in the JVM’s heap.
3. classname() : It is the constructor, which initializes the newly constructed object.

 ex:

 Employee e1 = new Employee();

 Note: For a class we can create multiple objects also.

 || DATE: 16-Aug-24 ||

/\*

 \* write a program to create a class Employee

 \* with attributes empno, ename and salary.

 \* In main method, create two objects for Employee

 \* class, display the employees details.

 \*/

**package** com.ait.oop;

**class** Employee {

 **int** empno;

 String ename;

 **double** salary;

 **void** display() {

 System.***out***.println("empno : " + empno);

 System.***out***.println("ename : " + ename);

 System.***out***.println("salary : " + salary);

 }

}

**public** **class** MainClass {

 **public** **static** **void** main(String[] args) {

 //create an object for Employee class

 Employee e1 = **new** Employee();

 //initalize the object with data

 e1.empno = 7788;

 e1.ename = "SCOTT";

 e1.salary = 6000.0;

 //call the method

 //syntax: objectname.methodname();

 e1.display();

 System.***out***.println("===================================");

 //create another object for Employee object

 Employee e2 = **new** Employee();

 //intialize the object with data

 e2.empno = 7201;

 e2.ename = "KING";

 e2.salary = 5000.0;

 //call the method

 e2.display();

 }

}

/\*

 \* write a program to create a class Player

 \* with attributes playerId, playerName and

 \* playerRank.

 \* Define a method to display the player details.

 \* In main method, create two objects for Player class,

 \* initialize the objects with data and display

 \*/

**package** com.ait.oop;

**class** Player {

 **int** playerId;

 String playerName;

 **int** playerRank;

 **void** display() {

 System.***out***.println("Player id : " + playerId);

 System.***out***.println("player name : " + playerName);

 System.***out***.println("player rank : " + playerRank);

 }

}

**public** **class** TestClass {

 **public** **static** **void** main(String[] args) {

 Player player1 = **new** Player();

 player1.playerId = 10101;

 player1.playerName = "JATIN";

 player1.playerRank = 4;

 player1.display();

 System.***out***.println("===================");

 Player player2 = **new** Player();

 player2.playerId = 10202;

 player2.playerName = "NEERAJ";

 player2.playerRank = 2;

 player2.display();

 }

}

getter and setter methods:

* Suppose, while creating a class, if don’t declare the variables as private, then they are directly accessible at outside of the class also.
* Suppose, another class, who is creating an object to my class is setting invalid data, then my program can get bugs.
* So, to disallow accessing the variables of my class directly, I should declare the variables as private.
* When we declare the variables as private, we should define public setter and getter methods to set the value and to retrieve the value of a variable.
* suppose, if you have a variable int xxx

then setter and getter methods can be created like below.

private int xxx;

public void setXxx(int xxx) {

 this.xxx = xxx;

}

public int getXxx() {

 return xxx;

}

/\*

 \* write a program to create a class Employee

 \* with attributes empno, ename and salary.

 \* Define setter and getter methods for the properties.

 \* In main method, create an object, call the setter

 \* methods to store the values, then display

 \*/

**package** com.ait.oop;

**class** Employee {

 **private** **int** empno;

 **private** String ename;

 **private** **double** salary;

 **public** **int** getEmpno() {

 **return** empno;

 }

 **public** **void** setEmpno(**int** empno) {

 **this**.empno = empno;

 }

 **public** String getEname() {

 **return** ename;

 }

 **public** **void** setEname(String ename) {

 **this**.ename = ename;

 }

 **public** **double** getSalary() {

 **return** salary;

 }

 **public** **void** setSalary(**double** salary) {

 **this**.salary = salary;

 }

 **void** display() {

 System.***out***.println("empno : " + empno);

 System.***out***.println("ename : " + ename);

 System.***out***.println("salary : " + salary);

 }

}

**public** **class** MainClass {

 **public** **static** **void** main(String[] args) {

 //create an object for Employee class

 Employee e1 = **new** Employee();

 e1.setEmpno(7788);

 e1.setEname("John");

 e1.setSalary(5000.0);

 e1.display();

 System.***out***.println("===================================");

 }

}

 constructor

 ------------

* Initializing an object means, assigning some values to the data members.
* Constructor is a special method in a class, which is used to initialize the data members, while creating an object.
* Suppose, if we don’t create a constructor in a class, then JVM will provide a default constructor and it will initialize the data members with default values, based on the data type of the data members.
* If you want to initialize the data members with meaningful data, then you have to define a constructor in your class.
* Constructor is a special method, because of the below reasons.
1. constructor does not contain return type.
2. constructor name must be same as the class name.
3. constructor is automatically executed while creating an object.

 for ex:

 class ClassA {

 int x; This class has no constructor,

 int y; so JVM will provide default

 } constructor.

 For ex:

 class ClassOne {

 int x; This class has a constructor.

 int y; So, JVM will not provide any

 ClassOne() { default constructor.

 x = 10;

 y = 20;

 }

 }

Types of constructors:

1. parameter-less constructor (or) constructor without arguments.
2. parameterized constructor (or) constructor with arguments.
* When we want to initialize the multiple objects with same data for the data members then we define parameter-less constructor.
* When we want to initialize the objects with different data for the data members then we define parameterized constructor.

For ex:

class ClassA {

 int x;

 int y;

 ClassA() { //parameter-less constructor

 x = 100;

 y = 200;

 }

 ClassA(int x, int y) { //parameterized constructor

 this.x = x;

 this.y = y;

 }

}

 || DATE: 19-Aug-24 ||

this keyword:

-------------

ex1:



* In the above example, the parameters names are a,b and attributes names are x and y in the constructor.
* The parameters names and attributes names are different. So, using this keyword is optional.

ex2:



* In the above example, the parameters names x,y and attributes names x,y are same.
* So, we have used this keyword, to distinguish the attribute names with parameter names.
* The keyword, this always refers the current object of a class.

ex3:



* In the above example, two objects are created for ClassA.
* when first object(ca) is initializing, this refers ca.
* when second object(ca2) is initializing, this refers ca2.

Access modifiers in Java:

* modifier is a word which updates the meaning of another word.
* In Java, we have 2 types of modifiers.
1. Access modifier
2. non-access modifier.
* In Java, we have 4 access modifiers.
1. private
2. default
3. protected
4. public
* Access modifier controls the visibility of a member of a class.

default:

 🡪 The default access modifier for a variable, or for a class, or for a constructor or for a method is default.

 🡪 To make a member as default, don’t write default before of the member.

 🡪 default members are visible with in the same package classes. But not visible at outside of the package.

 private:

* if we make a variable, or a constructor or a method as private, then it is visible only with in the same class.

protected:

* If we make a variable, or a constructor or a method as protected, then it is visible with in the same package and also in the child class of another package also.

public:

* If we make a variable, or a constructor or a method or a class as public, then it is visible with in the same package and also visible in other packages.

private < default < protected < public

 Note: At class-level(outer class), default and public access modifiers are allowed. private and protected are not allowed.

===========================================================

 || DATE: 23-Aug-24 ||

static method:

--------------

* A method is created to performe a well-defined task.
* a method can be static or instance method in a class.
* if a method performs a task by utilizing the data

of an object then we should define that method as an instatce method/non-static method.

* if a method performs a task by not depending/utilizing the data of an object then we should define that method as a static method.
* To make a method as static, we use static modifier in the method definition point.

ex:

 class Employee {

 private int empno;

 private double salary;

 Employee(int empno, double salary){

 this.empno = empno;

 this.salary = salary;

 }

 //instance method

 public void display() {

 S.o.p(empno + “ “ + ename);

 }

 //static method

 public static double calculateBonus(doule sal)

 {

 return sal \* 0.15;

 }

 }

 public class Main {

 p s v m(String[] args) {

 Employee e1 = new Employee(101, 5000.0);

 //display() is instance method

 //we can only call display() method

 //with object name.

 e1.display();

 //compile-time error

 Employee.display();

 //calculateBonus() is a static method

 //static method can be called with classname.

 //if we call with object name, compiler will

 //replace objectname with classname.

 double x = e1.calculateBonus(2000.0);

 //correct

 double y = Employee.calculateBonus(4000.0);

 }

}

static block:

------------

* The purpose of creating a static block is to initialize the static variables of a class.
* If we initialize the static variable in a constructor, then for every object creation of that class, constructor will be executed and it will re-initialize the static variable again and again.
* That’s why we use static block.
* static block will be executed for only once, when the class is loaded into JVM.
* In a class, we can define multiple static blocks also.
* If mulitple static blocks are defined then they are executed in the same order of the definitions.

ex:

 class A {

 private int x;

 private static int y;

 static { // static block

 y = 50;

 }

 A(int x) { //constructor

 this.x = x;

 }

 }

Q) what is the diff between static block and constructor?

A) 1. static block executes for only once at class loading time. But constructor executes for each object creation for a class.

 2. static block can only initialize static variables. But constructor can initialize both static and instance variables.

 3. this keyword is not allowed in static block. But it is allowed in constructor.

 ===================================================

 || DATE: 26 – Aug – 24 ||

/\*

 \* create a class called Booking with the below

 \* attributes/fields.

 \* bookingId

 \* bookingDate

 \* seatsRequired

 \* seatsAvailable

 \* Define methods to book/cancel the seats.

 \* In main method, create one or more objects for

 \* Booking class, invoke the methods.

 \*

 \*/

**import** java.time.LocalDate;

**import** java.util.Random;

**class** Booking {

 **private** **long** bookingId;

 **private** LocalDate bookingDate;

 **private** **int** seatsRequired;

 **private** **static** **int** *seatsAvailable*;

 **static** {

 *seatsAvailable* = 10;

 }

 Booking(LocalDate bookingDate, **int** seatsRequired) {

 **this**.bookingDate = bookingDate;

 **this**.seatsRequired = seatsRequired;

 }

 **public** **void** bookSeats() {

 **boolean** isBooked = **false**;

 **if** ( **this**.seatsRequired <= *seatsAvailable* ) {

 *seatsAvailable* = *seatsAvailable* - seatsRequired;

 isBooked = **true**;

 }

 **if** (isBooked) {

 Random random = **new** Random();

 **this**.bookingId = random.nextLong(28776199);

 System.***out***.println("Booking successful! Your booking id : "+ **this**.bookingId);;

 System.***out***.println("The seats remaining after booking : " + Booking.*fetchAvailableSeats*());

 System.***out***.println("Booking Date : " + **this**.bookingDate);

 System.***out***.println("Seats booked : " + **this**.seatsRequired);

 }

 **else** {

 System.***out***.println("Sorry, seats are not available!!");

 }

 }

 **public** **void** cancelSeats() {

 *seatsAvailable* = *seatsAvailable* + **this**.seatsRequired;

 System.***out***.println("seats are cancelled!!");

 System.***out***.println("The available seats after cancellation : " + Booking.*fetchAvailableSeats*());

 }

 **public** **static** **int** fetchAvailableSeats() {

 **return** *seatsAvailable*;

 }

}

**public** **class** MainClass {

 **public** **static** **void** main(String[] args) {

 //create Booking object

 Booking booking1 = **new** Booking(LocalDate.*now*(), 6);

 booking1.bookSeats();

 System.***out***.println("=====================================================");

 //creating another Booking object

 Booking booking2 = **new** Booking( LocalDate.*now*(), 7 );

 booking2.bookSeats();

 System.***out***.println("====================================================");

 booking1.cancelSeats();

 System.***out***.println("===================================================");

 booking2.bookSeats();

 }

}

output:

Booking successful! Your booking id : 16864654

The seats remaining after booking : 4

Booking Date : 2024-08-26

Seats booked : 6

=====================================================

Sorry, seats are not available!!

====================================================

seats are cancelled!!

The available seats after cancellation : 10

===================================================

Booking successful! Your booking id : 15416963

The seats remaining after booking : 3

Booking Date : 2024-08-26

Seats booked : 7

 Inner classes:

* If we create a class inside another class then it is called inner class.
* When we have related classes, and a class can’t work independently without the other class then

we can create that class in the other class.

 For ex:

 class Mobile {

 }

 class Sim {

 }

 The class Sim can’t work independently without Mobile class. So, here we can create Sim class inside Mobile class.

 class Mobile { // outer class

 //variables

 //methods

 class Sim { // inner class

 //variables

 //methods

 }

 }

* With inner classes, we can keep the related classes together at one place for better encapsulation, readability and maintianability.
* private variables of outer class are visible to the inner class, but private variables of inner class are not visible to the outer class.
* inner classes are 4 types.
1. non-static inner class
2. static inner class
3. local inner class
4. anonymous inner class

 ex1:

 class OuterClass {

 class InnerClass { //non-static inner class

 }

 }

ex2:

 class OuterClass {

 static class InnerClass { //static inner class

 }

 }

ex3:

 class OuterClass {

 void m1() {

 class InnerClass { //local inner class

 }

 }

 }

 Q) can we make inner class(static/non-static) as private?

 A) yes.

 Q) can we make local inner class as private?

 A) No.

 Q) can you make outer class as private?

 A) No.

 Q) can we make outer class as static?

 A) No.

 Q) If outer class and inner class has the same variable then which variable gets more priority?

 A) inner class variable gets priority.

 ex:

 **class** OuterClass {

 **private** **int** x = 10;

 **class** InnerClass {

 **private** **int** x = 20;

 **public** **void** m11() {

 System.***out***.println(x);

 }

 }

 **public** **void** m1() {

 InnerClass innerClass = **new** InnerClass();

 innerClass.m11();

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 OuterClass outerClass = **new** OuterClass();

 outerClass.m1();

 }

}

output: 20

Q) can we use inner class private variable in outer class?

A) yes. Even though it is private variable in the inner class but it’s scope/visibility is upto the outer class.

Q) Where do you create inner class object?

A) In outer class or at outside of the outer class also.

**class** OuterClass {

 **private** **int** x = 10;

 **class** InnerClass {

 **private** **int** x = 20;

 **public** **void** m11() {

 System.***out***.println(x);

 }

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 OuterClass outerClass = **new** OuterClass();

 //outerclassname.innerclassname objectname = outerclassobject.new innerclassname();

 OuterClass.InnerClass innerClass = outerClass.**new** InnerClass();

 innerClass.m11();

 }

}

Q) can a static inner class/nested class can use instance variables of outer class or not?

A) No. It can only use static variables of the outer class.

ex:

**class** OuterClass {

 **private** **static** **int** *x* = 10;

 **static** **class** InnerClass {

 **private** **int** y = 20;

 **public** **void** m11() {

 System.***out***.println(*x*);

 }

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 OuterClass outerClass = **new** OuterClass();

 //outerclassname.innerclassname objectname = new outerclassname.innerclassname();

 OuterClass.InnerClass innerClass = **new** OuterClass.InnerClass();

 }

}

 Inheritance

 ===========

* while creating a new class, suppose if some of the properties or methods are already exist in another class, then we create the new class by inheriting the properites and behaviour from the existing class. This is called inheritance.
* Inheritance is a process of creating new classes from the existing classes.
* The class who is inheritance the properties/behaviour from the existing class is called child class/sub class/derviced class. That existing is called parent class/super class/base class.
* we use extends keyword to create a child class from the parent class.

class ParentClass {

}

class ChildClass extends ParentClass {

}

* Inheritance improves developer’s productivity, provides code reusability, reduces redundency, reduces inconsistency and improves readability.

 Types of inheritance:

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance.



/\*

 \* create a parent class Camera with attributes

 \* brand, resolution, price

 \* create a child class DigitalCamera with attributes

 \* storage, sensorType

 \* In main method, create child class object and display

 \* the details.

 \*/

/\*

 \* The default parent class for any class in Java

 \* is java.lang.Object.

 \*/

**class** Camera { //parent class

 **private** String brand;

 **private** String resolution;

 **private** **double** price;

 //parameterized constructor

 **public** Camera(String brand, String resolution, **double** price) {

 **super**();

 **this**.brand = brand;

 **this**.resolution = resolution;

 **this**.price = price;

 }

 **public** **void** display() {

 System.***out***.println("Brand : " + brand);

 System.***out***.println("Resolution : " + resolution);

 System.***out***.println("Price : " + price);

 }

}

**class** DigitalCamera **extends** Camera { //child class

 **private** String storage;

 **private** String sensorType;

 //parameterized constructor

 **public** DigitalCamera(String brand, String resolution, **double** price, String storage, String sensorType) {

 **super**(brand, resolution, price);

 **this**.storage = storage;

 **this**.sensorType = sensorType;

 }

 **public** **void** printDetails() {

 display();

 System.***out***.println("Storage : " + storage);

 System.***out***.println("Sensor Type : " + sensorType);

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 //child class object

 DigitalCamera digitalCamera = **new** DigitalCamera("Canon", "100 MP", 200000.0, "5 GB", "CMOS");

 digitalCamera.printDetails();

 }

}

=================================================

super() call:

--------------

* super() call calls parent class constructor.
* super() call must be the first statement in the constructor.
* if you don’t write the super() call then java compiler will attach the super() call as a first statement by default.

ex1:

**class** A {

 A() {

 System.***out***.println("A : In parameter-less constructor");

 }

}

**class** B **extends** A {

 B() {

 System.***out***.println("B : In parameter-less constructor");

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 B bRef = **new** B();

 }

}

output:

 A : In parameter-less constructor

 B : In parameter-less constructor

ex2:

**class** A {

 A(**int** x) {

 System.***out***.println("A : In parameterized constructor(one parameter)");

 System.***out***.println("x = " + x);

 }

}

**class** B **extends** A {

 B() {

 **super**(10);

 System.***out***.println("B : In parameter-less constructor");

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 B bRef1 = **new** B();

 }

}

output:

 A : In parameterized constructor(one parameter)

 x = 10

 B : In parameter-less constructor

ex3:

**class** A {

 A(**int** x) {

 System.***out***.println("A : In parameterized constructor(one parameter)");

 System.***out***.println("x = " + x);

 }

}

**class** B **extends** A {

 B(**int** x) {

 **super**(x);

 System.***out***.println("B : In parameter-less constructor");

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 B bRef1 = **new** B(30);

 B bRef2 = **new** B(40);

 }

}

output:

A : In parameterized constructor(one parameter)

x = 30

B : In parameter-less constructor

A : In parameterized constructor(one parameter)

x = 40

B : In parameter-less constructor

================================================================

this() call :

-------------

* this() call calls the same class constructor
* this() call must be the first statement.
* if we don’t write this() call, then super() call will be added by the compiler.
* we can’t write both super() call and this() call at a time. Because, both calls must be the first statement in the constructor and it is not possible.

ex:

**class** A {

 A() {

 System.***out***.println("A : In parameter-less constructor");

 }

}

**class** B **extends** A {

 B() {

 **this**(100);

 System.***out***.println("B : In parameter-less constructor");

 }

 B(**int** a) {

 System.***out***.println("B : In parameterized constructor(one parameter)");

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 B bRef1 = **new** B();

 }

}

output:

A : In parameter-less constructor

B : In parameterized constructor(one parameter)

B : In parameter-less constructor

 || DATE: 30-Aug-24 ||

* In inheritance, we can store the child class object in the parent class reference variable. But we can’t store the parent class object in a child class reference variable.

ex:

 class ClassA {

 }

 class ClassB extends ClassA {

 }

 class Main {

 p s v m(String[] args) {

 ClassA ca = new ClassB(); //valid

 ClassB cb = new ClassA(); //error

 }

 }

* With parent class reference variable, we can only call parent class methods. But we can’t call child class methods.
* With child class reference variable, we can call both parent class methods and also child class methods.

ex:

 class ClassA {

 void m1() {

 Sysout(“parent class : m1()”);

 }

 class ClassB extends ClassA {

 void m2() {

 Sysout(“child class : m2()”);

 }

 }

 class Main {

 p s v m(String[] args) {

 ClassA ca = new ClassB();

 ca.m1(); //valid

 ca.m2(); //error

 ClassB cb = new ClassB();

 cb.m1(); //valid

 cb.m2(); //valid

 }

 }

Q) why can’t we implement multiple inheritance with classes?

A) because of ambibuity in calling parent class constructor and a parent class method.

ex1:

 class A { class B {

 A() { B() {

 } }

 } }

 class C extends A, B {

 C() {

 }

 }

 When child class, that is C class object is created, its constructor calls the parent class constructor. Because of 2 parent classes, the compiler gets an ambiguity. So, it is an error.

ex2:

 class A { class B {

 void m1() { void m1() {

 } }

 } }

 class C extends A, B {

 void m3() {

 super.m1();

 }

 }

* Here, child class is calling m1() method of the parent class. But both the parent classes have m1() method. So, the compiler gets ambiguity and it is an error.

/\*

 \* write a program to implement the following.

 \* create Vehicle class with attributes speed and fuelType.

 \* Define parameterized constructor and a display() method.

 \* create Car class from Vehicle with attributes numberOfDoors and

 \* transmissionType. Define parameterized constructor and a display()

 \* method.

 \* create ElectricCar class from Car with attributes batteryCapacity

 \* and chargingTime. Define parameterized constructor and a dispaly()

 \* method.

 \* In main class, create ElectricCar object, and call the display()

 \* method.

 \*/

**class** Vehicle {

 **private** **int** speed;

 **private** String fuelType;

 Vehicle(**int** speed, String fuelType) {

 **this**.speed = speed;

 **this**.fuelType = fuelType;

 }

 **void** display() {

 System.***out***.println("speed : " + speed);

 System.***out***.println("fuel type : " + fuelType);

 }

}

**class** Car **extends** Vehicle {

 **private** **int** numberOfDoors;

 **private** String transmissionType;

 Car(**int** speed, String fuelType, **int** numberOfDoors, String transmissionType) {

 **super**(speed, fuelType);

 **this**.numberOfDoors = numberOfDoors;

 **this**.transmissionType = transmissionType;

 }

 **void** display() {

 **super**.display();

 System.***out***.println("Numer of doors : " + numberOfDoors);

 System.***out***.println("Transmission type : " + transmissionType);

 }

}

**class** ElectricCar **extends** Car {

 **private** **int** batteryCapacity;

 **private** **int** chargingTime;

 **public** ElectricCar(**int** speed, String fuelType, **int** numberOfDoors, String transmissionType, **int** batteryCapacity,

 **int** chargingTime) {

 **super**(speed, fuelType, numberOfDoors, transmissionType);

 **this**.batteryCapacity = batteryCapacity;

 **this**.chargingTime = chargingTime;

 }

 **void** display() {

 **super**.display();

 System.***out***.println("Battery capacity : " + batteryCapacity);

 System.***out***.println("Charging time : " + chargingTime);

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 ElectricCar ec = **new** ElectricCar(80, "Petrol", 4, "Manual", 200, 2);

 ec.display();

 }

}

 Polymorphism

* Polymorphism means, multiple forms of doing a task.
* Polymorphism is of two types:
1. static/compile-time polymorphism
2. dynamic/runtime polymorphism.
* compile-time polymorphism can be implemented through method overloading.
* runtime polymorphism can be implemented through method overriding.

method overloading:

* method overloading means, defining the same method for multiple times in the same class, but with a difference in parameters.
* for example, in String class, substring() method is overloaded. Because, to return a portion of a string value, there are two ways. one way is with begin index and other way is with begin,end indexes.

substring(beginIndex)

substring(beginIndex, endIndex)

* In method overloading, the methods must contain the same name, but they must have the difference in parameters. The difference could be,
1. number of parameters, or
2. data type of the parameters, or
3. order/sequence of the parameters.

 ex:

 **class** Recharge {

 **void** doRecharge(**long** mobileNumber, **double** amount) {

 System.***out***.println("doRecharge(mobileNumber, amount)");

 }

 **void** doRecharge(**long** mobileNumber, **double** amount, String couponCode) {

 System.***out***.println("doRecharge(mobileNumber, amount, couponCode)");

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 Recharge recharge = **new** Recharge();

 recharge.doRecharge(9008007001L, 399.0);

 recharge.doRecharge(6309087652L, 778.0, "FEST200");

 }

}

* In the above Recharge class, we have doRecharge() method is defined for 2 times.
* one time with 2 parameters and other time with 3 parameters. So, it is method overloading.

ex:

 public class Login {

 public boolean login(String username, String password)

 {

 }

 public boolean login(String emailId, String password)

 {

 }

 }

* Here, login() method is defined for 2 times. But there is no difference in number of parameters, or data type of the parameters or in the order of the parameters. So it is not method overloading. It generates a compile-time error.

 ex:

 class ClassA {

 void m1(int x, double y) {

 }

 void m1(double a, int b) {

 }

 }

* Here, m1() method is defined for 2 times. Both methods have 2 parameters, but their order is different. So, it is overloading.

 ex:

 class Repository {

 Employee find(int id) {

 }

 void find(int id) {

 }

 }

* Here, find() method is defined 2 times. But there is no difference in parameters. So, it is not method overloading. It’s a compile-time error.
* method overloading doesn’t depends on return type of a method.

ex:

 class A {

 void m1(int x) {

 }

 }

 class B extends A {

 void m1(int x, int y) {

 }

 }

* Here, child class has two m1() methods. One is inherited from the parent class and the other one is defined in the child class.
* one m1() has one parameter and the other m1() method has two parameters. So, it is method overloading.

 Please add notes of date 2-sep-2024 here

 || DATE: 3 – SEP – 2024 ||

static method overriding:

* we can’t override static method of a parent class in the child class.
* If you create a static method in the child class, whose signature is matching with parent static method, then it is said to be method hiding.
* If you want to get an error, while trying override a static method of the parent class then use @Overrid annotation.

ex1:

**class** ClassA {

 **static** **void** sayHello() {

 System.***out***.println("sayHello() : ClassA");

 }

}

**class** ClassB **extends** ClassA {

 **static** **void** sayHello() { // no error, it’s not overriding

 System.***out***.println("sayHello(): ClassB");

 }

}

ex2:

**class** ClassA {

 **static** **void** sayHello() {

 System.***out***.println("sayHello() : ClassA");

 }

}

**class** ClassB **extends** ClassA {

 @Override

 **static** **void** sayHello() { //error

 System.***out***.println("sayHello(): ClassB");

 }

}

@Override : This annotation, is used to tell the compiler that

 this method is overriding a parent class method.

 So, compiler checks whether the method signature is

 matching with the parent class method or not.

 If not, then compiler will generate an error.

 This annotation can be added only before a method.

**class** ClassA {

 **void** doCaliculationForTax() {

 System.***out***.println("doCalicationForTax() : ClassA");

 }

}

**class** ClassB **extends** ClassA {

 @Override

 **public** **void** doCaliculationForTax() {

 System.***out***.println("doCaliculationForTax(): ClassB");

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 ClassB classB = **new** ClassB();

 classB.doCaliculationForTax();

 }

}

Q1) can we overload a constructor?

1. yes

Q2) can we override a constructor?

1. No

Q3) can we overload a instance method?

1. yes

Q4) can we override an instance method?

1. yes

Q5) can we overload a static method?

1. yes

Q6) can we override a static method?

1. No

Q7) can we overload a non-static method in the same class?

1. Yes

Q8) can we overload a non-static method in the child class?

1. Yes.

Q9) can we overload a static method in child class?

1. No.

Q10) can we overload a private non-static method in the child class?

1. No

Q11) can we overload a private static method in the same class?

1. Yes.

//method overriding demo program

**class** BankAccount {

 **private** **long** accountNumber;

 **private** **double** balance;

 **public** BankAccount(**long** accountNumber, **double** balance) {

 **super**();

 **this**.accountNumber = accountNumber;

 **this**.balance = balance;

 }

 **public** **long** getAccountNumber() {

 **return** accountNumber;

 }

 **public** **double** getBalance() {

 **return** balance;

 }

 **public** **double** calculateInterest() {

 **return** balance \* 0.05;

 }

 **public** **void** display() {

 System.***out***.println("account number : " + accountNumber);

 System.***out***.println("balance : " + balance);

 }

}

**class** SavingsAccount **extends** BankAccount {

 **public** SavingsAccount(**long** accountNumber, **double** balance) {

 **super**(accountNumber, balance);

 }

 @Override

 **public** **double** calculateInterest() {

 **return** getBalance() \* 0.15;

 }

}

**class** CurrentAccount **extends** BankAccount {

 **public** CurrentAccount(**long** accountNumber, **double** balance) {

 **super**(accountNumber, balance);

 }

 @Override

 **public** **double** calculateInterest() {

 **return** getBalance() \* 0.25;

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 System.***out***.println("Savings account details ....");

 SavingsAccount sAccount = **new** SavingsAccount(21564329L, 6000.0);

 sAccount.display();

 System.***out***.println("interest amount : " + sAccount.calculateInterest());

 System.***out***.println("=======================");

 System.***out***.println("Current account details ......");

 CurrentAccount cAccount = **new** CurrentAccount(39877190L, 14000.0);

 cAccount.display();

 System.***out***.println("interest amount : " + cAccount.calculateInterest());

 }

}

HAS-A relationship:

* Inheritance represents IS-A relationship between the class.
* extends keyword can be read as IS-A

for ex:

 class Mobile extends Product {

 }

 Mobile IS-A product

* HAS-A relationship means, create a reference variable of one class in the another class as a data member.

ex:

 class ClassA {

 ClassB classB; // HAS-A

 ClassA() {

 classB = new ClassB();

 }

 }

* Here, ClassA is called as a dependent class and ClassB is called as a dependency class.
* HAS-A relationship can be either Aggregation or Composition.
* Aggregation is a weak bonding/relation between a depedent class and a dependency class.
* For example.

class Policy {

 Customer policyHolder;

 Policy() {

 policyHolder = new Customer();

 }

}

* Here, Policy is dependent class and Customer is dependency class.
* Customer class can work independently without Policy class also. This is called weak bonding. So, the relationship is called Aggregation.
* Composition is a strong bonding/relation between dependent and dependency classes.

ex:

class Person {

 Passport passport;

 Person() {

 passport = new Passport();

 }

}

* Here, Person is a dependent class and Passport is a dependency class.
* Passport can’t work independently without a Person. So, it is called composition.

final keyword:

* final keyword can be used with the following.
1. variable
2. method
3. class
* final indicates finalized/fixed.
* If you declare a variable as final then it becomes like a constant in a program.
* If you declare a variable as final then it is object level constant.
* If you declare a variable as static final then it is class level constant.

ex1:

 class BankAccount {

 private final long accountNumber;

 private double balance;

 }

* here, accountNumber is declared as final, because its value is fixed for an account object. It is object level constant.

ex2:

 class MyMath {

 public static final doule PI = 3.142;

 }

* here, PI is declared as a static final, because its value is fixed for all instances of the MyMath class. It is class level constant.
* It is recommended to write the variable name in uppercase, if a variable is static and final.
* If a variable is final, we can define a getter method, but we can’t define a setter method.
* If you declare a method as final then it can’t be overridden in the child classes.

class Order {

 public final void processOrder() {

 //logic

 }

}

class AmazonFulfilledOrder extends Order {

}

class MerchantFulfilledOrder extends Order {

}

* Here, processOrder() is a common logic for the two types of orders and they are not allowed to change the processOrder() logic. So, we declare this method as final in the Order class.
* private methods, static methods and final methods can’t be overridden.
* final methods can be overloaded.

ex:

 class Product {

 public final double getPrice(int productId)

 {

 //logic

 }

 public final double getPrice(String category, String brand, String name, String size) {

 //logic

 }

 }

 abstract classes

 ==================

* In Java, abstraction principle can be implemented by creating abstract classes or interfaces.
* If you create a class and given to another class then you are showing the entire information of that class to the other class. So, there is no abstraction here.
* Basically, A project doesn’t need to have all the OOPS principles. Based on the project requirements,

we will implement the required principles.

* In a project, we call some classes as a family of classes, if they have parent-child relationship.
* While creating a parent class, if it knows that some functionality is required to the child classes, but the implementation of that functionality should be different from one child class to another class, then parent class declares that method as abstract method.

For example:

 abstract class Shape {

 abstract double calculateArea();

 void changeColor(String color) {

 //logic

 }

 }

 class Circle extends Shape {

 @Override

 double calculateArea() {

 //some logic

 }

 }

 class Rectangle extends Shape {

 @Override

 double calculateArea() {

 //some logic

 }

 }

* In the above, we have Shape(parent class) and

Circle, Rectangle child classes.

* The class Shape knows that, calculateArea() functionality is required for the child classes, but the implementation should be differred. That’s why it has declared calculateArea() method as an abstract method.
* In the Shape class, we have one abstract method and one concrete method.
* A concrete method means, it has method definition.
* If a class contains atleast one abstract method, then you must declare that class as abstract class.
* abstract denotes not fully implemented.
* abstract is a non-access modifier, which can be used with a method or a class.
* Suppose, if a child class is not overriding abstract method of the parent class then we have to declare the child class also as abstract class.

example:

**abstract** **class** ClassA {

 **int** x;

 **int** y;

 ClassA(**int** x, **int** y) {

 **this**.x = x;

 **this**.y = y;

 }

 **abstract** **void** m1();

}

**class** ClassB **extends** ClassA {

 ClassB() {

 **super**(10, 20);

 }

 @Override

 **void** m1() {

 System.***out***.println("overridden m1() in : ClassB");

 System.***out***.println("x = " + x);

 System.***out***.println("y = " + y);

 }

}

**public** **class** MainClass {

 **public** **static** **void** main(String[] args) {

 //for abstract class we can't create an object

 //ClassA ca = new ClassA();

 ClassB cb = **new** ClassB();

 cb.m1();

 }

}

Q) can we define a constructor in abstract class?

A) Yes.

Q) what is the use of the constructor in abstract class?

A) To initialize the data members of the abstract class.

Q) can we create object for abstract class?

A) No.

Q) can we create object for subclass of abstract class?

A) Yes. When we create child class object, its constuctor invokes the abstract class constructor.

Q) I have 2 abstract methods in abstract class, but child class is overriding only one abstract method. Will I get an error in the child class?

A) Yes. To solve the error, either we have to override the both abstract methods or else declare child class as abstract class.

Q) can we declare a class as abstract class without abstract methods?

A) Yes.

Q) When can we create abstract class without abstract methods?

A) 1. when we want disallow object creation for a class.

 2. when we want to tell the other developers that our class is not a fully implemented logic.

//abstract class example program

**abstract** **class** InsurancePolicy {

 **double** basePremium;

 InsurancePolicy(**double** basePremium) {

 **this**.basePremium = basePremium;

 }

 **public** **double** calculatePremium() {

 **return** basePremium + calculateRiskFactor();

 }

 **public** **abstract** **double** calculateRiskFactor();

}

**class** HealthPolicy **extends** InsurancePolicy {

 **int** age;

 **boolean** isSmoker;

 HealthPolicy(**double** basePremium, **int** age, **boolean** isSmoker) {

 **super**(basePremium); //calls parent class constructor

 **this**.age = age;

 **this**.isSmoker = isSmoker;

 }

 @Override

 **public** **double** calculateRiskFactor() {

 **double** riskFactor = 0; //local variable

 **if** ( age >=50 && age < 60 ) {

 riskFactor = basePremium \* 0.25;

 }

 **else** **if**( age >= 60 ) {

 riskFactor = basePremium \* 0.40;

 }

 **else** {

 riskFactor = 0;

 }

 **if**(isSmoker) {

 //add 15% extra for a smoker

 riskFactor += basePremium \* 0.15;

 }

 **return** riskFactor;

 }

}

**class** VehiclePolicy **extends** InsurancePolicy {

 **int** mfgYear;

 VehiclePolicy(**double** basePremium, **int** mfgYear) {

 **super**(basePremium);

 **this**.mfgYear = mfgYear;

 }

 @Override

 **public** **double** calculateRiskFactor() {

 **if** ( mfgYear >= 2022 ) {

 **return** basePremium \* 0.10;

 }

 **else** {

 **return** basePremium \* 0.35;

 }

 }

}

**public** **class** MainClass {

 **public** **static** **void** main(String[] args) {

 HealthPolicy hPolicy = **new** HealthPolicy(12999.0, 55, **true**);

 **double** finalPermiumToPay = hPolicy.calculatePremium();

 System.***out***.println("Final premium amount to pay for the health policy : " + finalPermiumToPay);

 VehiclePolicy vPolicy = **new** VehiclePolicy(21788.0, 2019);

 **double** finalAmountToPay = vPolicy.calculatePremium();

 System.***out***.println("Final amount to pay for the vehicle policy : " + finalAmountToPay);

 }

}

 Interfaces

 //include DATE: 9-9-24 notes here

 || DATE: 10-9-24 ||



PaymentProcessor.java

**public** **interface** PaymentProcessor {

 String processPayment(**double** amount);

 **boolean** paymentStatus(String txId);

}

PaypalProcessor.java

**import** java.util.Random;

**public** **class** PaypalProcessor **implements** PaymentProcessor {

 @Override

 **public** String processPayment(**double** amount) {

 **if** ( amount <= 0 ) {

 **return** "ppnull101";

 }

 **else** {

 **return** "pp"+**new** Random().nextInt(10000);

 }

 }

 @Override

 **public** **boolean** paymentStatus(String txId) {

 **if** ( txId.startsWith("ppnull"))

 **return** **false**;

 **else**

 **return** **true**;

 }

}

StripeProcessor.java

**import** java.util.Random;

**public** **class** StripeProcessor **implements** PaymentProcessor {

 @Override

 **public** String processPayment(**double** amount) {

 **if**( amount <= 0 ) {

 **return** "spnull101";

 }

 **else** {

 **return** "sp"+**new** Random().nextInt(10000);

 }

 }

 @Override

 **public** **boolean** paymentStatus(String txId) {

 **if** ( txId.startsWith("spnull"))

 **return** **false**;

 **else**

 **return** **true**;

 }

}

Main.java

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 PaymentProcessor paypal = **new** PaypalProcessor();

 **double** amount = 6000.0;

 **if** ( paypal.paymentStatus(paypal.processPayment(amount)) ) {

 System.***out***.println(amount + " : paid through paypal");

 }

 **else** {

 System.***out***.println(amount + " : is invalid.");

 }

 String line = "=".repeat(40);

 System.***out***.println(line);

 PaymentProcessor stripe = **new** StripeProcessor();

 amount = 9000;

 **if** ( stripe.paymentStatus(stripe.processPayment(amount))) {

 System.***out***.println(amount + " : paid through stripe");

 }

 **else** {

 System.***out***.println(amount + " : is invalid");

 }

 }

}

parameter passing in Java:

* Parameter passing is of 2 types.
* 1. pass-by-value
* 2. pass-by-reference
* In pass-by-value, the changes made to the formal parameter will not refect to the actual parameter.
* primtive types are pass-by-value.
* In pass-by-reference, the actual and formal parameters will point to the same object.
* If any changes are made by the formal parameter will reflect in the actual parameter also.
* If a new object is assigned to the formal parameter, then the changes made by the formal parameter will not reflect in the actual parameter. In this case, it is pass-by-value only.



 packages

 ----------------

* packages are created in an application, to organize the code in a good manner and also to avoid name conflicts.
* In package, we store a group of related classes together.
* Every package is a folder in the project.
* a package can have sub-packages also.
* So, a package can have classes, interfaces and sub-packages







